Added a **User-Agent** that mimics a real browser,reducing the chance of being blocked or flagged.

async def fetch(session, url, params=None):

Makes an asynchronous HTTP GET request using an aiohttp session.

Includes: params: query parameters for the URL (like q=search+term)

Headers: to simulate a browser timeout: sets max wait time for response (30 seconds) async - Allows multiple web requests to run concurrently (non-blocking), which speeds up scraping multiple pages.

async def fetch_page_ddg/bing – fetches information from web pages.

Fetch HTML:

The function sends a request to DuckDuckGo/Bing with the query "Python programming".

It retrieves the HTML content of the search results page.

Parse HTML:

The HTML content is parsed using BeautifulSoup.

Extract Results:

The function extracts the title and URL of each search result.

***Limit Results**:*

If total_results_to_fetch is 10, the function stops after extracting 10 results.

***Return Results**:*

The results are returned to the caller for further processing.

async def get_all_text_from_url(url):

Extract and clean all text content from a URL->it fetches and extracts readable text from a webpage effectively.

Sends a GET request to the given url with headers to mimic a browser. Parses the HTML using BeautifulSoup.

Removes unwanted tags (<script>, <style>) that don't contain useful content. Extracts visible text, removes extra whitespace. Returns a clean, human-readable string of text.

async def split_text_into_chunks(text, chunk_size):

Split text into chunks of approximately equal size Input: A long block of text and a chunk_size (usually in characters).

Output: A list of chunks, each containing complete sentences and approximately chunk_size characters.

Logic: Splits the text into sentences using a regular expression that preserves sentence-ending punctuation.

Iteratively adds sentences to a chunk until adding the next one would exceed the target chunk size. Starts a new chunk and repeats.

async def process_text_content(texts, chunk_size)

Uses asyncio.get_event_loop() + run_in_executor to: Run split_text_into_chunks() concurrently for each text input.

It splits the text at each sentence (by .).

It adds sentences to the current chunk until adding another one would exceed the chunk_size. Then it starts a new chunk.

async def query_ollama_llm(prompt)

Queries Ollama's local LLM llama 3.2 to generate a response based on a prompt.

Sends POST request to /api/generate with prompt.

async def get_embeddings_from_ollama(text_chunks)

Gets vector embeddings for each text chunk using Ollama's local embedding model **(nomic-embed-text**).

For each chunk: If non-empty, sends a POST request to Ollama's /api/embeddings. Appends the embedding or a zero-vector if it fails.

ex - { "model": "nomic-embed-text", "prompt": "we will win." } Output: We get back an embedding, which is a list of float numbers (e.g, a 768-dimensional vector): [0.00234, -0.0317, 0.1075, ..., 0.0541]

async def query_vector_store

function is responsible for querying the **FAISS index** to retrieve the top-k most relevant results based on the query embedding

The query_embedding (a 768-dimensional vector) is converted into a NumPy array with the correct shape and data type (float32).

FAISS requires the input to be in this format for similarity search.

The query embedding is compared with all embeddings in the FAISS index.

The top-k indices and distances are returned.

The results are formatted into a list of dictionaries, each containing:

title: The title of the webpage.

url: The URL of the webpage.

chunk: The text chunk.

Score: The similarity score.

async def fetch_and_process_data(search_query):

A ClientSession is created to manage HTTP requests.
Search results are fetched for up to 3 pages (adjustable).
Results are stored in the results list.
Each text is split into smaller chunks (e.g., 1000 characters each).
Each chunk is converted into an embedding using Ollama's embedding model.
The metadata (title, URL, chunk, embedding) is stored in the data list.
The metadata is saved to a CSV file (search_data.csv).
The embeddings are stored in a FAISS index (faiss_index.index) for fast similarity search.